Inherited Attributes

Inherited attributes can be computed from the values of attributes at siblings and parent of that node.

(or).

Inherited Attribute for a nonterminal B, at a parse tree Node N is defined by the Semantic rule associated with production at the parent of N. This production must have B in the body. Inherited attribute at Node N is defined by attribute values at N's parent, N itself and N's siblings.

Example

Following SDD computes terms like 3*5, and 3*5*7.

Production	Semantic rules.
1) $T \to FT'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \to *FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$.
3). $T' \to \epsilon$	$T'.syn = T'.inh$
4). $F \to digit$	$F.val = digit.lexval$.

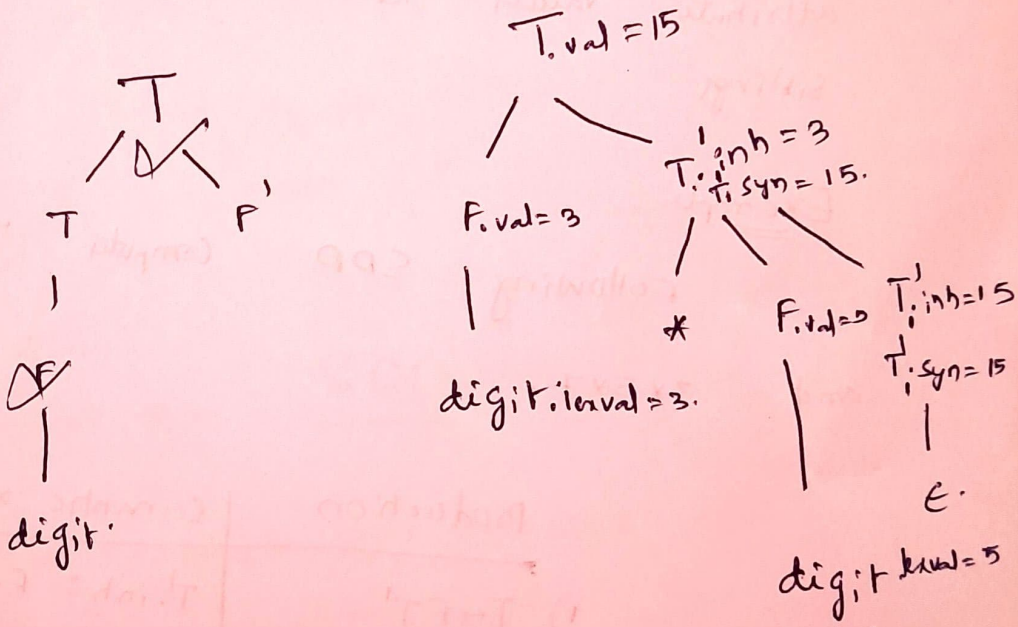
The top down parse starts from production

$T \to FT'$

F generates digit 3, * is generated by T'. Left operand 3 appears in a different Parse Tree from *, so an inherited Attribute is needed to pass the Operand to Operator.

Each Non Terminal T and F has Synthesized Attribute Val, digit has Synthesized attribute lexval. T' has Two Attributes, an inherited Attribute inh, and Synthesized Attribute Syn.

The Annotated parse Tree can be given as



The Semantic rules are applied as follows.

left most leaf in parse Tree, labeled digit. has attribute value = 3, which is supplied by lexical Analyzer, It is passed to its parent. F → digit, The semantic rule with this production defines F.val = digit.lexval, which Equals to 3.

At the second child of the root, The inherited Attribute $T'.inh$ is defined by rule $T'.inh = F.val$ of production 1, hence $T'.inh = 8$. The production at this node is $T' \rightarrow *FT_1'$ (subscripts are used to distinguish between two nodes of T').

The inherited attribute of $T_1'.inh$ is defined by rule $T_1'.in = T'.inh * F.val$ of production 2. With $T'.inh = 3$ $F.val = 5$, we get $T_1'.inh = 15$.

At the lower node $T' \rightarrow \epsilon$, semantic rule is $T'.syn = T'.inh$, which defines $T_1'.syn = 15$. The synthesized Attribute $T'_{syn} = 15$ is passed to root T_1 where $T.val = 15$.

Evaluation Order for SDD's.

Dependency graphs are useful tool for determining the evaluation order for determining the evaluation order for the attribute instances in a given parse Tree. Annotated parse Tree shows the values of attributes, a dependency graphs helps us determine how these values are computed.

Dependency Graph.

Dependency graph shows the flow of information among attribute instances in a parse Tree. An edge from one instance to other instance means, The value of first needs to compute the second. Edges express constraints implied by semantic rules.

→ For each parse Tree Node X, dependency graph has node for each attribute associated with X

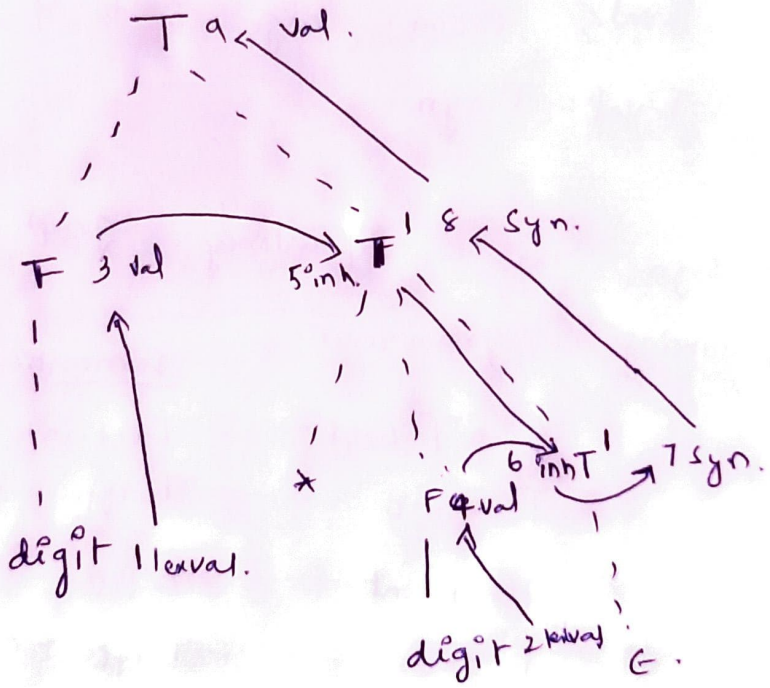
For example

→ Consider following production and semantic rule

Production	Semantic Rule.
$E \rightarrow E1 + T$	$E.val = E_1.val + T.val$

In parse Tree of This production, Every Node E, has Two children E and T in body. The Synthesized attribute val at head E is computed by using values of Two children labeled E and T. As shown below. For Convention, We show parse Tree as dotted lines, while The edges of dependency graph are shown by directed solid Arrows.

The Completed dependency Graph previous example of 3 * 5, can be given as follows. The Nodes of dependency graph, represented by the numbers 1 to 9, corresponds to attributes in annotated parse Tree.



Node 1 and 2 represents attribute lexval associated with two leaves labeled digit. Node 3 and 4 represents attribute val, associated with the two nodes labeled F. The edges from 1 to 3, and 2 to 4 result from semantic rule F.val = digit.lexval.

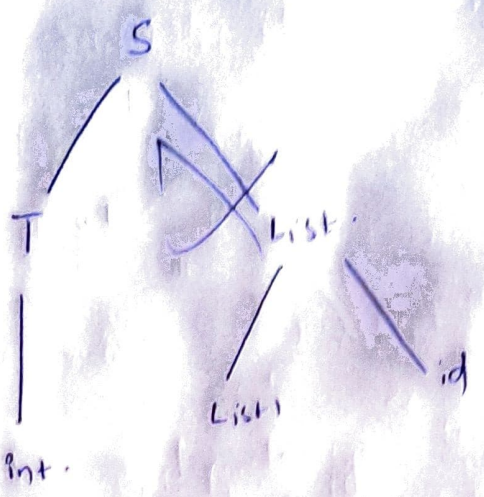
Node 5, 6 represent inherited attribute $T'.inh$. The edge From 3 to 5, is from semantic rule $T'.inh = F.val$ The edge from 5 to 6, is from ~~semantic rule~~ and 4 to 6 is from Semantic rule $T'.inh = T'.inh \times F.val$, Node 7, and 8 are Synthesized Attributes, 6 to 7 is computed from $T'.syn = T'.inh$, and, 7 to 8 is computed from $T'.syn = T_1'.syn$.

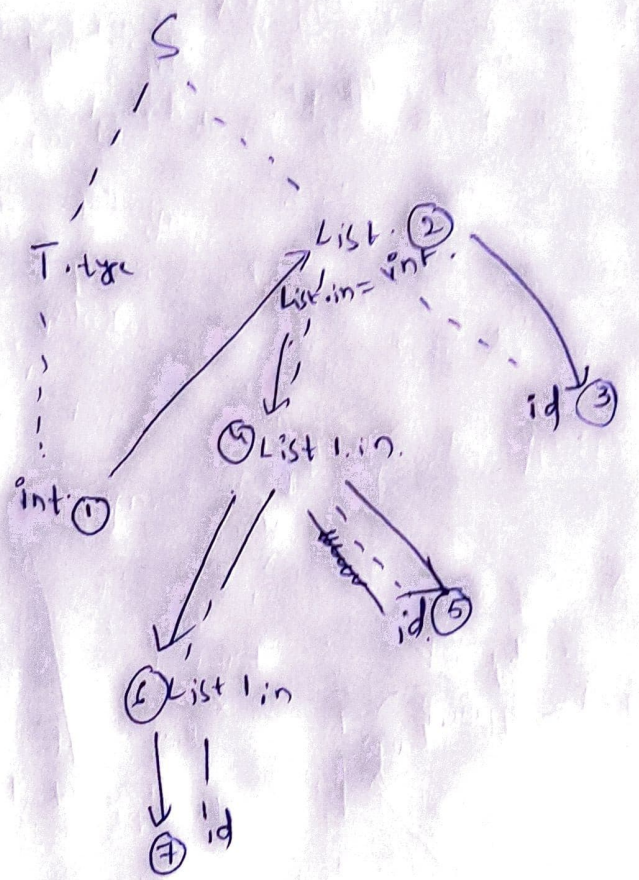
Finally Node 9 is represents Synthesized Attribute $T.val$, From edge 8 to 9, From semantic rule $T.val = T'.syn$.

Ex 2 :- Design the dependency graph for the following grammar

Semantic Actions.	
$S \rightarrow T\ List$	$List.inh = T.type$.
$T \rightarrow int$	$T.type = integer$.
$T \rightarrow Float$	$T.Type = Float$
$T \rightarrow char$	$T.type = char$.
$T \rightarrow double$	$T.type = double$.
$List \rightarrow List_1, id$	$List_1.inh = List.inh$ $Enter\_type(id.entry, List.in)$
$List \rightarrow id$	$Enter\_type(id.entry, List.in)$.

Draw dependency graph for int a, b, c.


S
T
List
List1
id
int


S
T.type
List
List.in = int
id
int
List1.in
id
List1.in
id

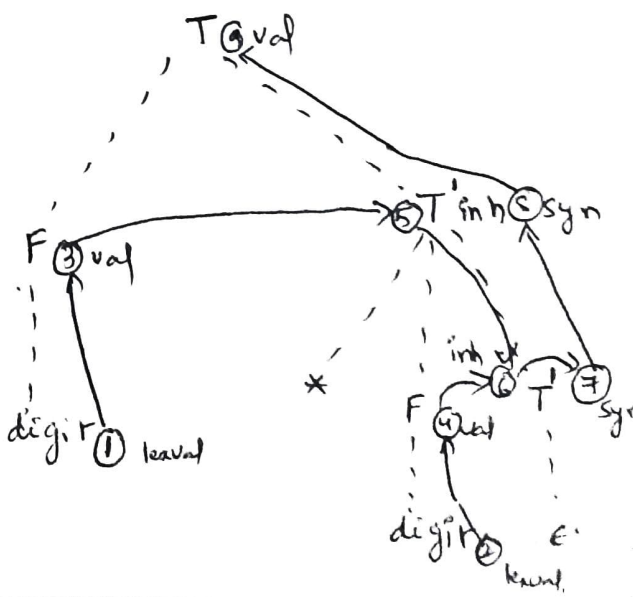
Ordering the Evaluation of Attributes

The Dependency graph displays possible orders in which we can evaluate the Attributes at various nodes of a parse Tree. If dependency graph contains edge from node M to node N, The Attribute corresponding to M must be evaluated before Attributes of N. The order of evaluation of Sequence of Nodes $N_1, N_2, \ldots N_k$, such that there is edge of dependency graph from N_i to N_j, then $i < j$. Such ordering consists of Linear order, and called as topological sort of the graph.

If there is any cycle in the graph, then there are no topological sorts, then there is no way to evaluate the SDD on this parse Tree. If there is no cycle, then there is always atleast one topological sort.

Example :- In previous dependency graph, With nodes 1, 2, ... 9, every edge of the graph goes from a node to higher-numbered Node.

The topological sort is 1, 3, 5, 2, 4, 6, 7, 8, 9



S- Attributed Definitions.

It is very difficult to see parse cycles in Parse Tree dependency Graph. Some classes of SDD's do not permit dependency graphs with cycles. The first class is defined as

→ An SDD is S-attributed if every attribute is Synthesized.

When SDD is S-attributed we can evaluate attributes in any bottom up order of the nodes of the parse Tree. These attributes are evaluated using post order traversal of parse Tree.

L- Attributed Definitions.

The second class of SDD's is called as L-attributed Definitions. The idea behind this is that between attributes in a production body, dependency graph edges can go from left to right but not from right to left. Each Attribute must be either.

1. Synthesized or.
2. Inherited. attributes must be at head

> 3. Inherited / Synthesized attributes must be located to the left of Symbol Xi.

S- Attributed Definition is also L- Attributed Definition

Ex:- The previous SDD is L-Attributed, For eg in it consider the productions, and semantic rules For inherited Attribute

Production	Semantic Rule
$T \to F T'$	$T'.inh = F.val$
$T' \to *FT_1'$	$T_1'.inh = T'.inh \times F.val$

⇒ $T'.inh$ is inherited using F.val, and F appears on left of T' in production body as required

⇒ $T_1'.inh$ is inherited using $T'.inh$, associated with head, and F, where F appears to the left of T_1' in production body.

In each of these cases, the rules use the info'n from above or from left, and remaining Attributes are Synthesized. Hence The SDD is L-attributed.

Semantic Rules With Controlled Side Effects.

Translations involves side effects. With SDD's we make balance b/w Attribute Grammars and Translations. The side effect may be like printing the result in Calculator

Following is a SDD for simple desk calculator

Production	Semantic Rules
1) $L \to E\ n$	L.val = E.val
2) $E \to E_1 + T$	E.val = E_1.val + T.val
3) $E \to T$	E.val = T.val
4) $T \to T_1 * F$	T.val = T_1.val * F.val
5) $T \to F$	T.val = F.val
6) $F \to (E)$	F.val = E.val.
7) $F \to digit$	F.val = digit.lexval

Concidee as an incidental side effect, The above SDD's first production is modified as

Production	Semantic Rule.
$L \to E$	Print (E.val)

When above semantic rule exicutes which prints E.val, will create a dummy Synthesized Attribute L.val (head of the production)

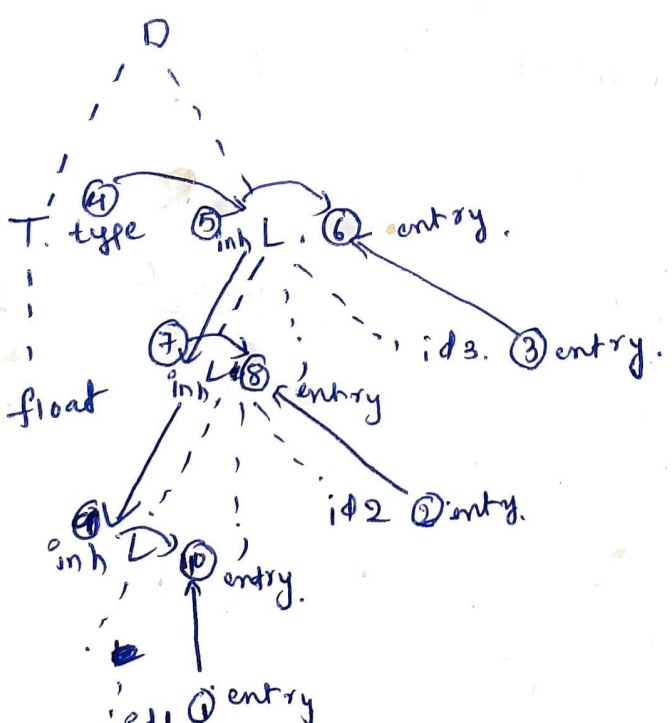
Ex:- Following SDD, has a Declaration D, consists of basic Type T followed by L of identifier.

Production	Semantic Rule.
1) D → T L	L.inh = T.type.
2) T → int	T.type = integer.
3) T → float.	T.type = float.
4) L → L1. id	L1.inh = L.inh addType (id.entry, L.inh)
5) L → id	addType (id.entry, L.inh)

The production 4 and 5, has a function addType which is called with two arguments

1. id.entry → lexical value points to symbol table object.
2. L.inh → Type assigned to every identifier on list.

The dependency graph for float id1, id2, id3; is as follows



The Nodes 6, 8, 10 are the dummy Attributes, they represent Application of function AddType to a Type, and One of these entry values.

Application of Syntax Directed Translation.

The main application of SDD is to create a Syntax Tree. Compiler uses Syntax Tree as intermediate representation, SDD turns input into a Tree. To generate intermediate code compiler walks through Syntax Tree

Construction of Syntax Tree.

Syntax tree is an abstract representation of a construct. In Syntax tree a node's children represents meaningful components of the construct. For ex, A Syntax tree node representing an expression $E_1 + E_2$, has label +, and two children representing subexpressions E_1 and E_2.

The nodes of a Syntax Tree is implemented by Objects with suitable number of fields. Each Object has 'Op' field, that is the label of the node. Additional Fields in a Object are as follows

* If the Node is a leaf, an additional field has lexical value for that leaf. A function Leaf (Op, Value) creates a leaf object. If a node is viewed as a record, then Leaf return pointer to new record for that leaf.

→ If the Node is an interior node, there will be as many additional fields as the node has children in Syntax Tree. A construction Function Node creates Two or more ~~operands~~ arguments : Node (op, c1, c2, . . . ck), creates an Object with first field Op, and k additional fields for K children c1 . . . ck.

Ex:- Consider following S-attributed definition which constructs a Syntax Tree using binary Operators + and −

production	Semantic Rule.
1) E → E1 + T	E.node = new Node('+', E1.node, T.node)
2) E → E1 − T	E.node = new Node('−', E1.node, T.node)
3) E → T	E.node = T.node
4) T → (E)	T.Node = E.Node.
5) T → id	T.node = new Leaf (id, id.entry)
6) T → num	T.node = new Leaf (num, num.val)

All nonterminals have one Synthesized Attribute node.

→ When production E → E1 + T is used The rule creates a node with '+', for op and Two children E1.node, and T.node. for Subexpressions

→ The Second production has similar rule.

Pointer to leaf of a is also value of E.node

→ production 3, E→T, does not create any node E.node is same as T.Node

→ Similary production 4, T→(E), also does not create any Node, The Value of T.Node is same as E.Node.

→ The last two productions has single terminal On right, for which Suitable Node "Leaf" will be created, which becomes Value of T.Node.

→ Now consider, construction of a-4+c. The Nodes of a Syntax Tree is shown as a ~~Tree~~ Record with first field Op. Syntax Tree Edges are shown as solid lines. The underlining Parse Tree which is Optional may be shown as dotted lines. The dashed line represents Values of E.node and T.node, Each dashed line points to appropriate Syntax Tree Node.

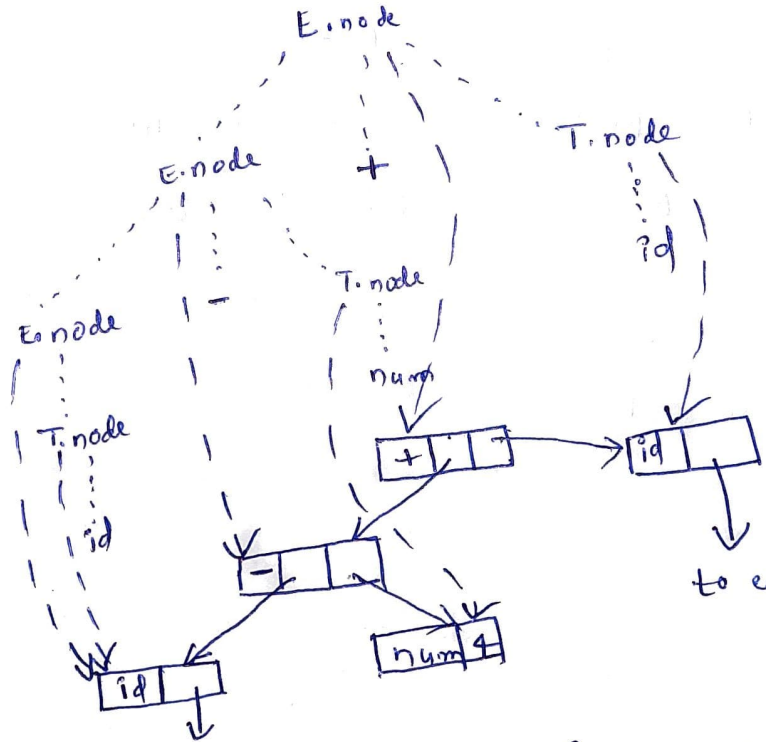
→ The Syntax Tree is constructed as follows

→ Bottom We see leaves for a, 4, c Constructed by Leaf. The entry id.entry points to symbol table and num.value is constant. The leaves (pointers to them) becomes ~~becom~~ Value of T.node at Three parse Tree Nodes T. According to rule 5 and 6. According to rule 3 pointer to leaf of a is also Value of E.node, leftmost

→ Rule 2 causes us to create a new node with Op equals to '-' and pointers to two leaves. The rule 1 produces the root node of Syntax Tree by combing '-' with third leaf

The pointers in post order Traversal can be given as (step in construction of Syntax tree a-4+c)

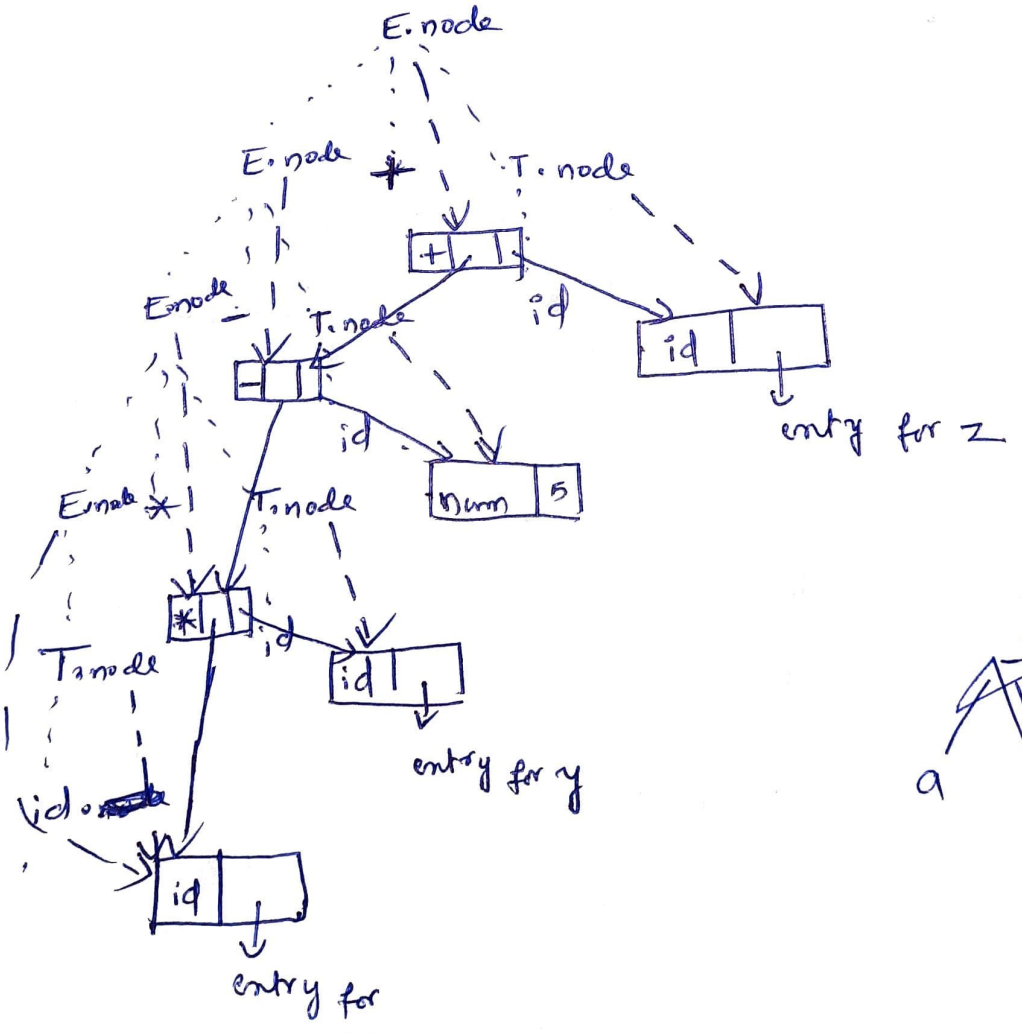
1) p_1 = new leaf (id, entry-a)
2) p_2 = new leaf (num, 4)
3) p_3 = New Node ('-', p_1, p_2)
4) p_4 = new Leaf (id, entry-c)
5) p_5 = new Node ('+', p_3, p_4)

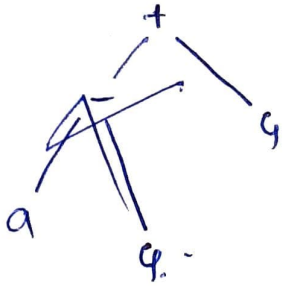


Syntax tree for a-4+c.

Ex2 Construct Syntax Tree for x*y-5+z. From following SDD

Production rule	Semantic Rules
$E \to E_1 + T$	E.node = new node ('+', E_1.node, T.node)
$E \to E_1 - T$	E.node = new node ('-', E_1.node, T.node)
$E \to E_1 * T$	E.node = new node ('*', E_1.node, T.node)
$E \to T$	E.node = T.node
$T \to id$	T.node = new leaf (id, id.entry)
$T \to num$	T.node = new leaf (num, num.value)

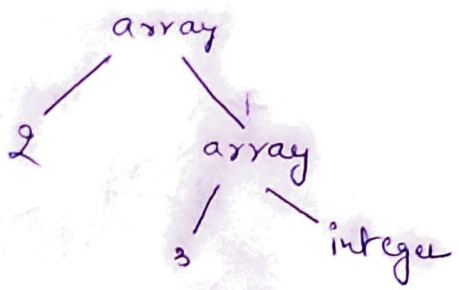




The Structure of a Type.

Inherited Attributes are useful when the Structure of parse Tree differs from the Syntax of input. Attributes are used to carry info from one part of parse Tree to another

Ex:- in C, The type int[2][3], can be read as 'array of 2 arrays of 3 integers' The Corresponding type expression is array(2, array(3, integer)) is represented by following Tree.



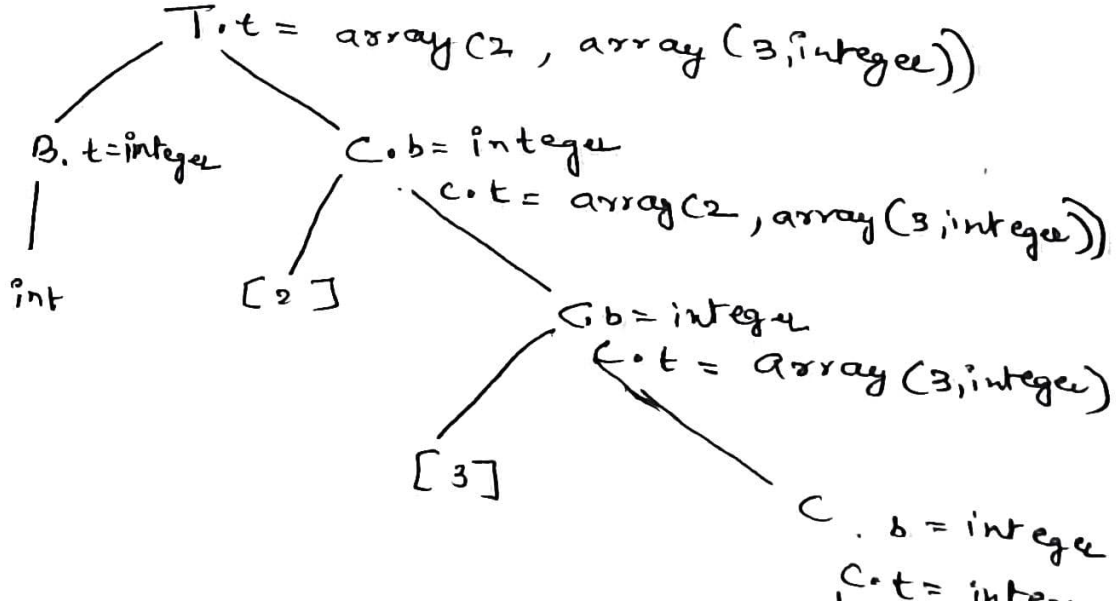
The Operator array takes two parameters, the number and type.

Following SDD generates either a basic type or array type

Production	Semantic Rules.
$T \rightarrow BC$	$T.t = C.t$ $C.b = B.t$
$B \rightarrow int$	$B.t = integer.$
$B \rightarrow float$	$B.t = float.$
$C \rightarrow [num]\ C_1$	$C.t = array(num.val, C_1.t)$ $C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b.$

From above SDD, non terminal T generates either a basic type or an array type. Non Terminal B generates basic types int and float. T generates basic Type when T derives BC, and C derives ϵ, Otherwise C generates array components consisting of sequence of integers, surrounded by brackets.

The Non Terminal B and T have Synthesized Attributes t representing type. C has two attributes an inherited attribute b and synthesized attribute t. The inherited Attribute passes basic type down the Tree, and Synthesized Attribute t passes the result up the root for T→BC, nonterminal C inherits the type from B, using inherited Attribute C.b. At the right most node for C, The production C→ϵ, so C.t equals to C.b. The semantic rules for the production C→[num]C1, form C.t by applying operator array to the operands num and C1.t



Syntax-Directed Translation Schemes (SDT)

SDT schemes are complementary notion to SDD. All the Applications of SDD can be implemented using SDT

Syntax Directed Translation (SDT) scheme is a context free Grammar with program fragments embedded within production bodies. These program fragments are known as semantic actions and can appear at any position within production body. We place Curly Braces around actions, if Thu braces comes under grammer Symbols, then we use quotes.

SDT can be implemented by first building a parse Tree and then performing actions in a left-to-right depth first order, i.e, during pre order traversal. Typically SDT's are implemented during parsing, without building a parse Tree.

Post Fix Translations.

The simplest SDD implementation includes in bottom up parse, and S-attributed Grammer. We can construct an SDT, in which each action is placed at the end of the production and is executed along with reduction of the body to the head of that production. SDT's with all

actions at the right end production bodies are called postFix SDT's.

Ex:- Folowing SDT implements Desk calculator The Actions are exact counter parts of semantic rules This Grammar is LR, and the SPD is S-attributed, These actions can be performed along with reduction steps of the parser.

$L \to E\ n$ { print (E.val); }

$E \to E_1 + T$ { E.val = E1.val + T.val; }

$E \to T$ { E.val = T.val; }

$T \to T_1 * F$ { T.val = T_1.val X F.val; }

$T \to F$ { T.val = F.val; }

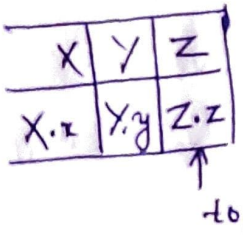
$F \to (E)$ { F.val = E.val; }

$F \to digit$ { F.val = digit.lexval; }

Parser-Stack implementation of postFix SDT's.

Post Fix SDT's can be implemented during LR parsing by creating the actions When Reduction occurs. The Attributes of each grammer symbol can be put on stack, where they can be found during the reduction. The Attributes are placed along with grammar symbols on the stack.

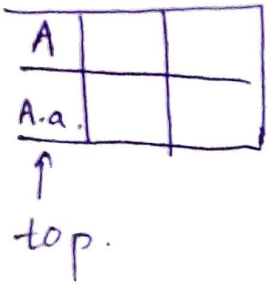
Following figure represents parser stack contains records with a field for a grammar symbol and below it a field for an attribute. For a production $A \rightarrow XYZ$, we first add X, Y, then Z on to stack along with attributes $X.x, Y.y, Z.z$ If all these attributes are synthesized, and actions occur at the end of production, We can compute attributes for head when we reduce body to head. If we reduce $A \rightarrow XYZ$, When we have attributes on the stack as below



State/Grammar Symbol

Synthesized attributes.

This can be reduced by removing all attributes on top of stack, i.e., and replacing head A, previous position of X.



Ex:- The Desk Calculator SDT can be modified to show manipulate the parser stack explicitely. Stack manipulation is done automatically by the parser.

Production	Actions
$L \to En$	{ print(stack[top-1].val); top = top-1; }
$E \to E_1 + T$	{ stack[top-2].val = stack[top-2].val + stack[top].val; top = top-2; }
$E \to T$	
$T \to T_1 * F$	{ stack[top-2].val = stack[top-2].val X stak[top].Va top = top-2; }
$T \to F$	
$F \to (E)$	{ stack[top-2].val = stack[top-1]. top = top-2; }
$F \to digit$	

→ top is a cursor to the top of stack

Stack[top] refers top record on the stack

Stack[top-i] points to record below it. Each record has a field ~~had~~ val, which holds attributes value.

For eg, The second production E → E1+T, We go two positions below the ~~stack~~ top to get value of E1, and we find value of T on top, The resulting sum ~~placed~~ is placed at the head E after the reduction, The Three top most symbols are replaced by one. After computing E.val, we pop off two symbols, and the record E.val is placed on top.

In production E → T, no action is necessary because length of the stack does not change The value of T.val becomes E.val. Same for F → (E), but slightly different due to parenthesis

SDT's With Actions inside Productions.

An Action can be placed at any position within the body of predection. Action is executed immediately after all symbols to its left are processed.

For exv, if we have production

$B \rightarrow X \{a\} Y$, The action a is done after we recognize X (if it is a terminal) or all terminals derived from X if it is a non terminal

→ If bottom-up parse is used, action 'a' is performed when X appears on top of stack.

→ If parse is top down, action "a" is performed before we attempt to expand the occurance of Y.

Ex:- Following SDT prints the prefix form of an expression

1). $L \rightarrow E$ n.

2). $E \rightarrow \{$ print ('+'); $\}$ $E_1 + T$

3). $E \rightarrow T$

4). $T \rightarrow \{$ Print ('*'); $\}$ $T_1 * F$

5). $T \rightarrow F$.

6) $F \rightarrow (E)$

7) $F \rightarrow$ digit { print digit.lexval); }.

SDT can be implemented as follows.

1. Ignore the Actions, Parse the input and produce a parse Tree as a result.
2. Then Examine Each intermediate Node N, for production say $A \to \alpha$. Add Additional childrens to N for actions in α. So children in N from left to right have exactly the same Symbols and actions of α
3. Perform pre order traversal of tree, if a node with action is visited perform that Action.

Ex:- Following Tree represents parse Tre for Expression 3*5+4 With Actions inserted. If we visit the nodes in pre order, we get prefix form of the expression +*354.